CSE502: Foundations of Parallel Programming

Lecture 08: Task Scheduling Paradigms

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture: **Library** Based Thread Pool Runtime

```
volatile boolean shutdown = false;
void init_runtime() {
  int size = thread_pool_size();
  for(int i=1; i<size; i++) {
    pthread_create(worker_routine);
  }
}
```

```
void worker_routine() {
  while( !shutdown ) {
    find_and_execute_task();
  }
}
```

```
void find_and_execute_task() {
   //pop_from_runtime is thread-safe
   task = pop_task_from_runtime();
   if(task != NULL) {
      execute_task(task);
      free(task);
      lock_finish();
      finish_counter--;
      unlock_finish();
   }
}
```

```
volatile int finish_counter = 0;
void start_finish() {
  finish_counter = 0; //reset
}
```

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
void async(task) {
  lock_finish();
  finish_counter++;//concurrent access
  unlock_finish();
  // copy task on heap
  void* p = malloc(task_size);
  memcpy(p, task, task_size);
  //thread-safe push_task_to_runtime
  push_task_to_runtime(&p);
  return;
}
```

```
void end_finish() {
  while(finish_counter != 0) {
    find_and_execute_task();
  }
}
```

```
void finalize_runtime() {
   //all spinning workers
   //will exit worker_routine
   shutdown = true;
   int size = thread_pool_size();
   // master waits for helpers to join
   for(int i=1; i<size; i++) {
     pthread_join(thread[i]);
   }
}
```

Today's Class

- Lab-1 solution
- Task scheduling paradigms
  - Work-sharing scheduling
  - Work-stealing scheduling

How to Push/Pull Tasks in Runtime ?

- push_task_to_runtime()
- pop_task_from_runtime()

We saw the use of these two runtime APIs in Lecture 07

Data-structures for storing tasks in a thread pool based runtime plays a very important role in determining the scalability and performance of the runtime

How to Push/Pull Tasks in Runtime ?

- push_task_to_runtime()
- pop_task_from_runtime()

- Two widely used task scheduling techniques
  - Work-sharing
    - OpenMP parallel for loops
  - Work-stealing
    - OpenMP tasking pragmas, Cilk, X10, HClib, Habanero-Java

Today's Class

- Lab-1 solution
- Task scheduling paradigms
  - Work-sharing scheduling
  - Work-stealing scheduling

Work–Sharing

Shelf of Files

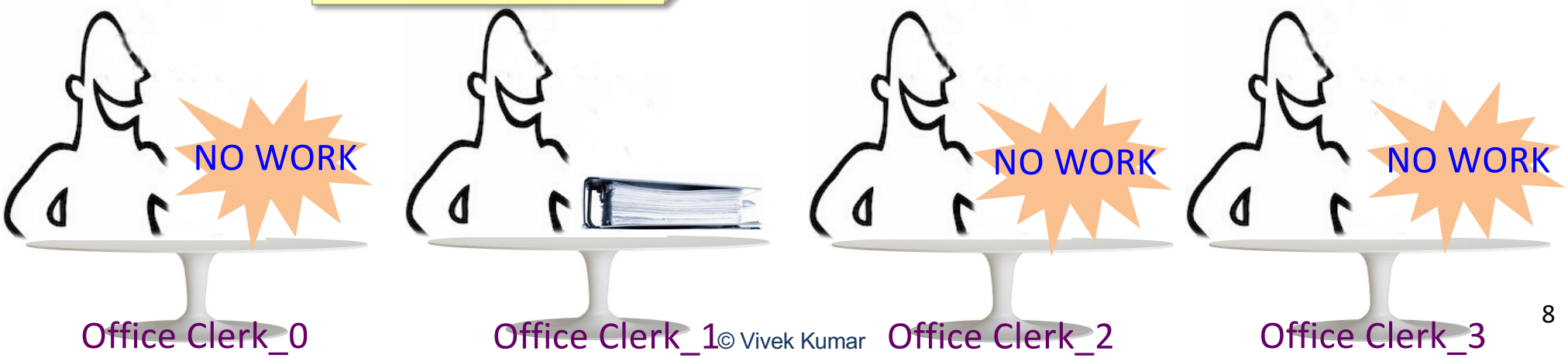
```
volatile boolean shutdown = false;
void init_runtime() {
  int size = thread_pool_size();
  for(int i=1; i<size; i++) {
    pthread_create(worker_routine);
  }
}
```

Work–Sharing

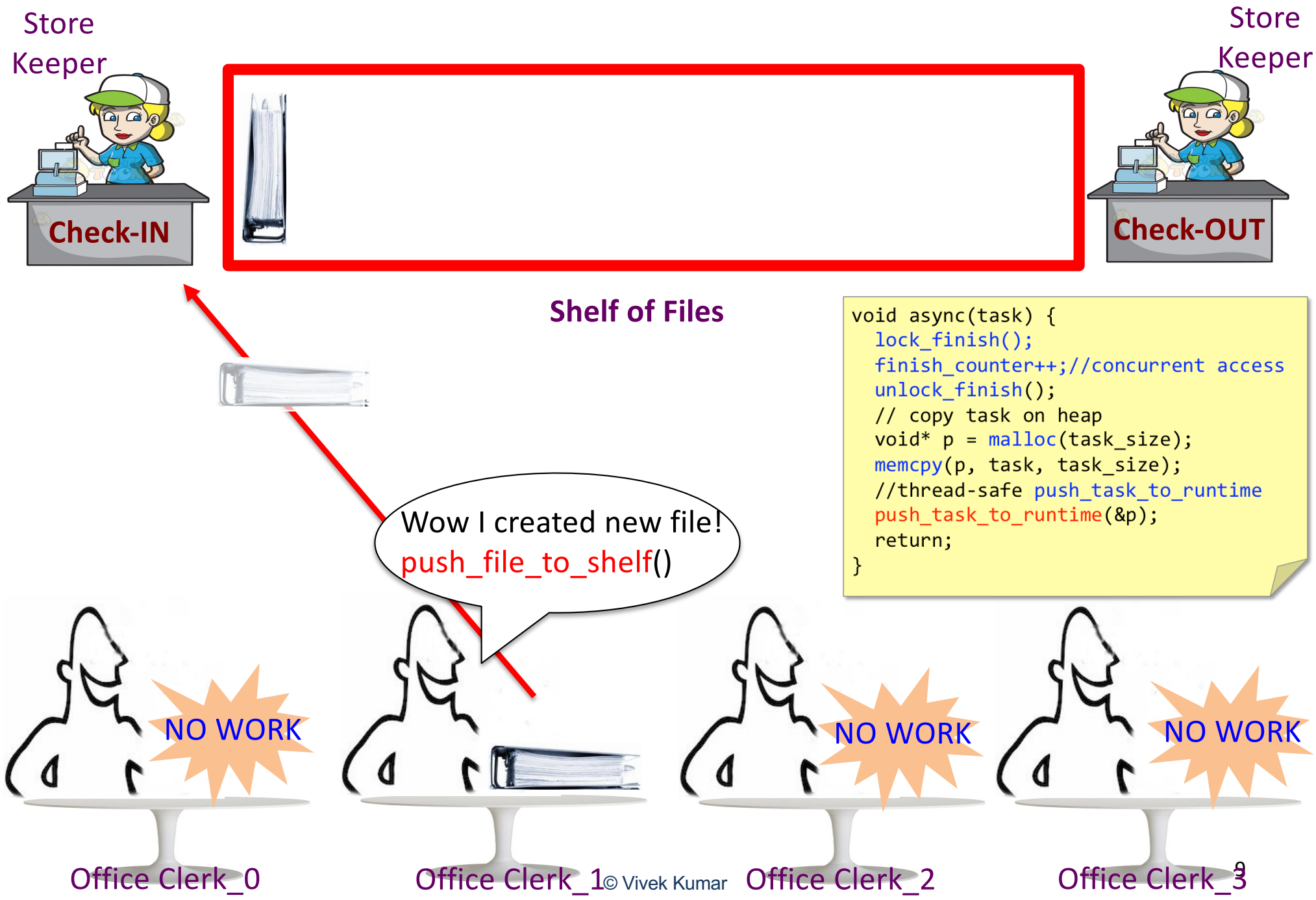
Shelf of Files

```
#include <runtime-API.h>
main() {
  init_runtime();
  start_finish();
    async (S1);
    S2;
  end_finish();
  finalize_runtime();
}
```

```
void worker_routine() {
  while( !shutdown ) {
     find_and_execute_task();
  }
}
```

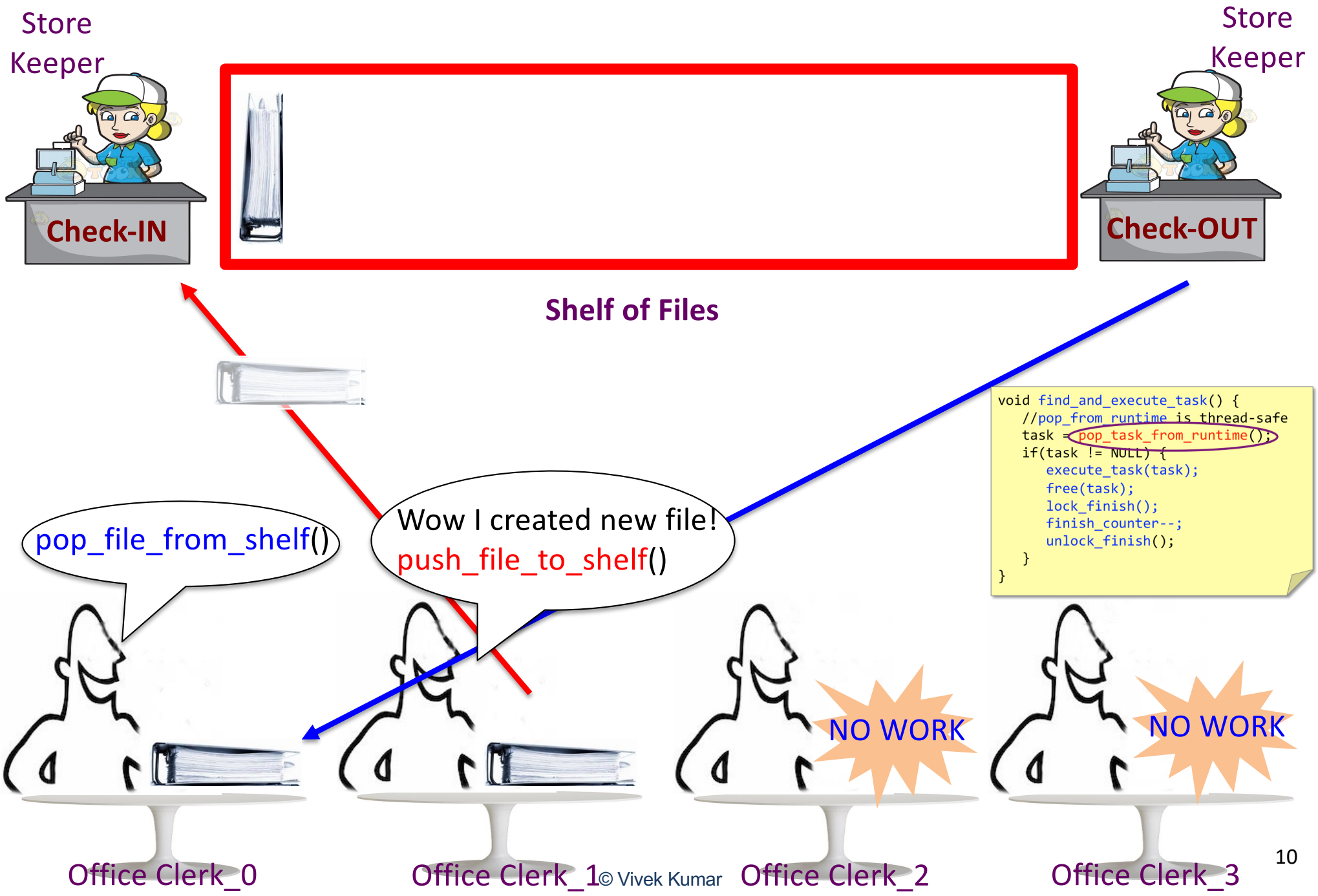


Work–Sharing

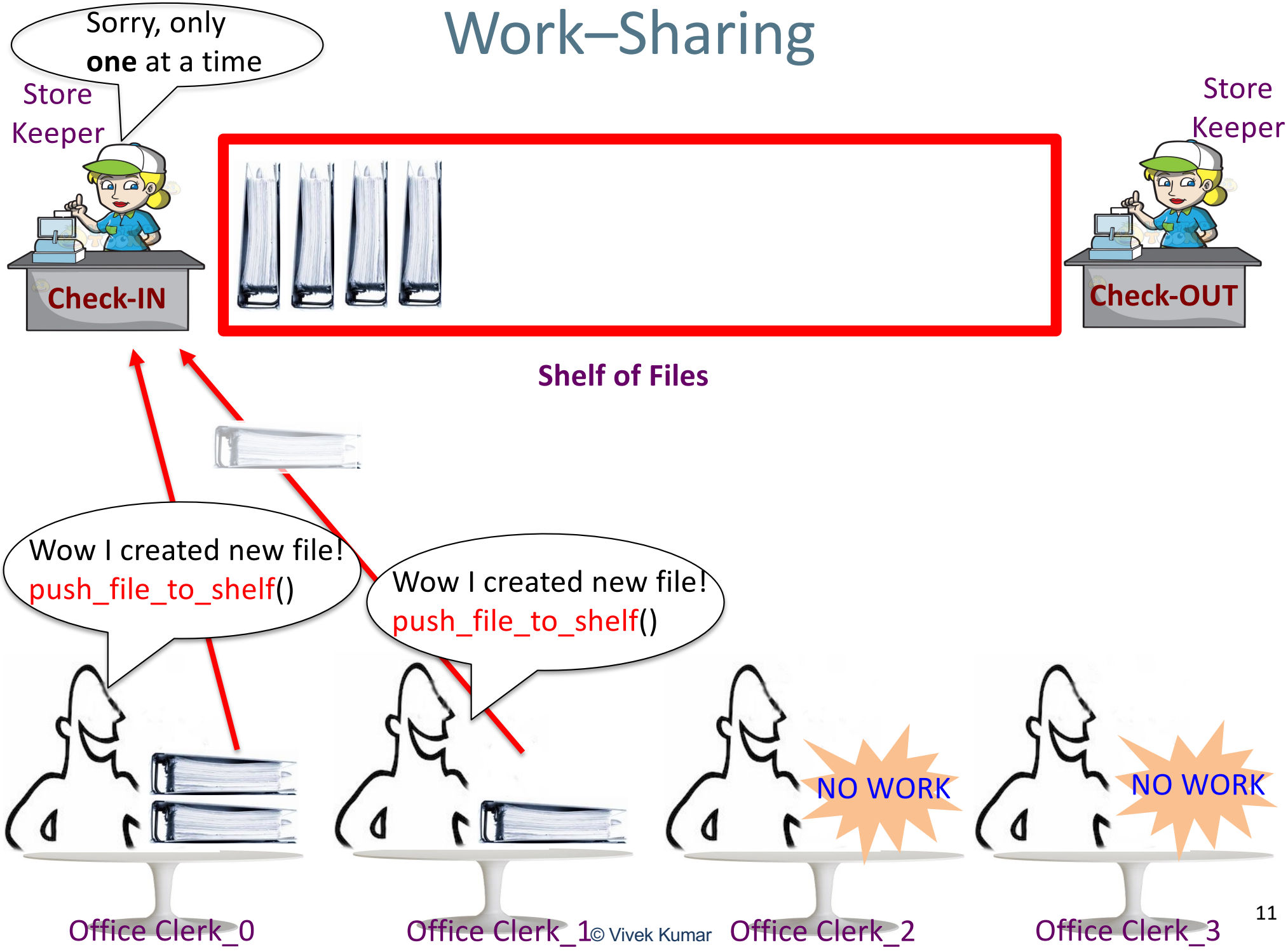



```
void async(task) {
  lock_finish();
  finish_counter++;//concurrent access
  unlock_finish();
  // copy task on heap
  void* p = malloc(task_size);
  memcpy(p, task, task_size);
  //thread-safe push_task_to_runtime
  push_task_to_runtime(&p);
  return;
}
```

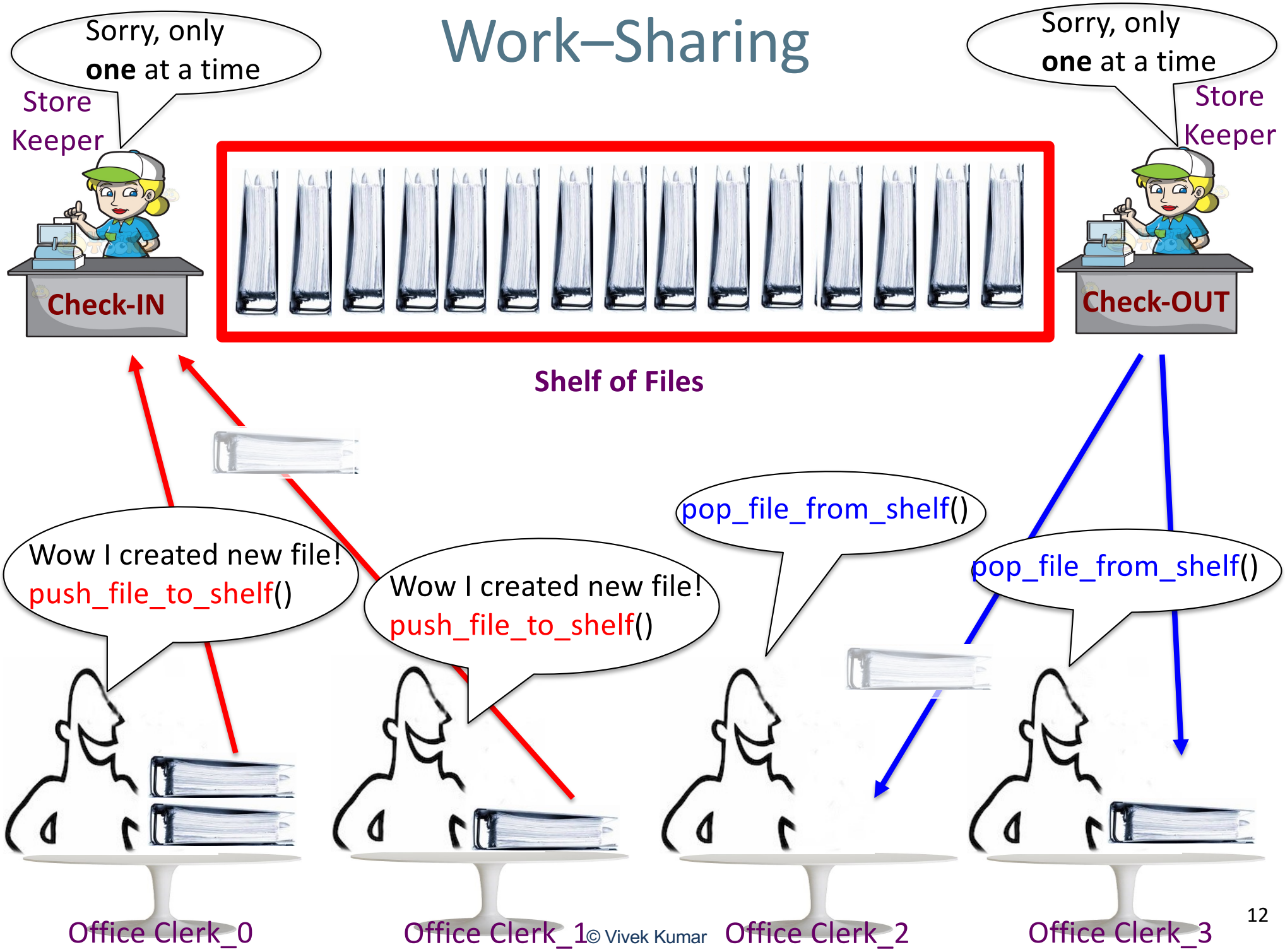
Work–Sharing



Work–Sharing



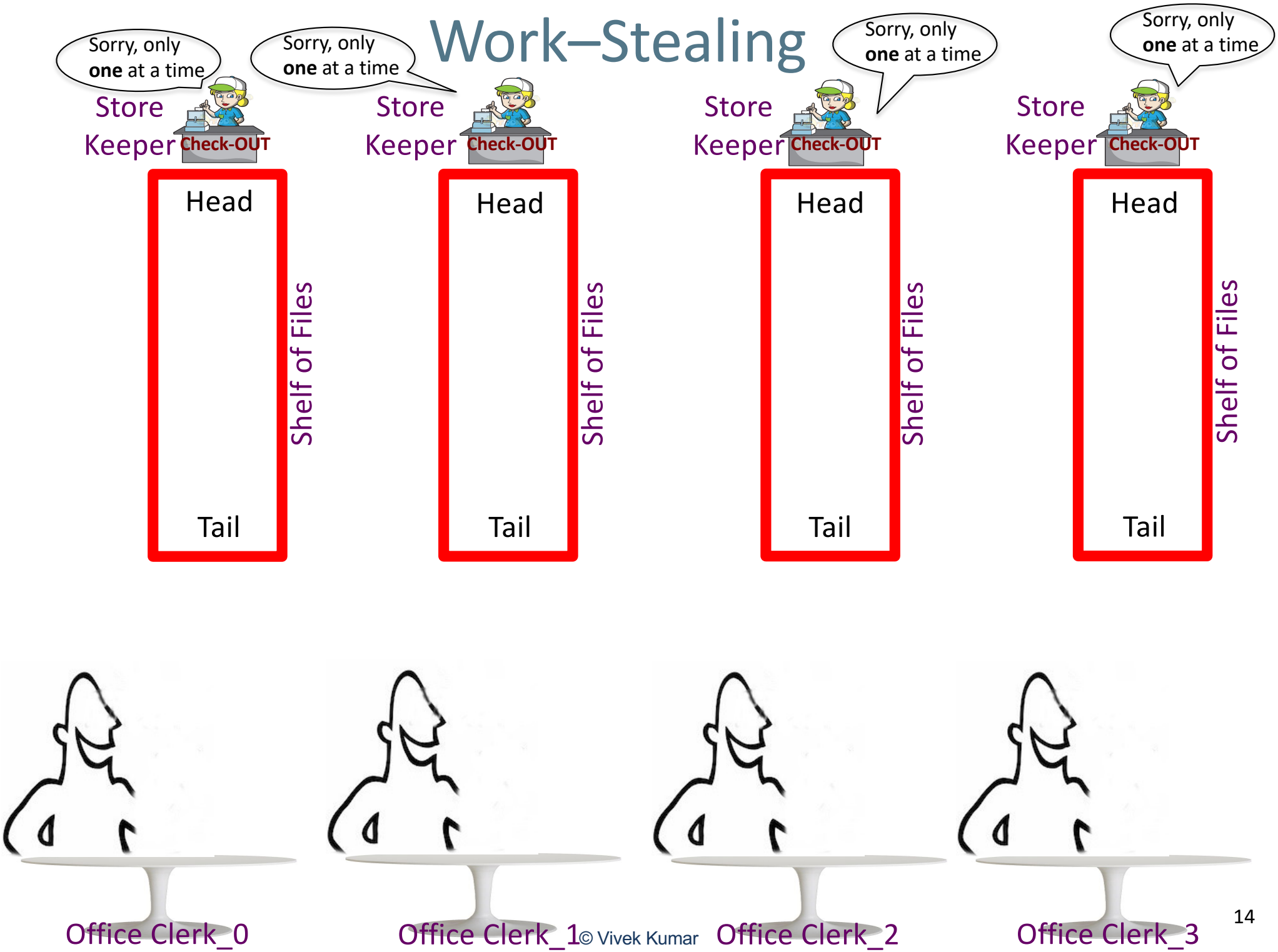
Work–Sharing



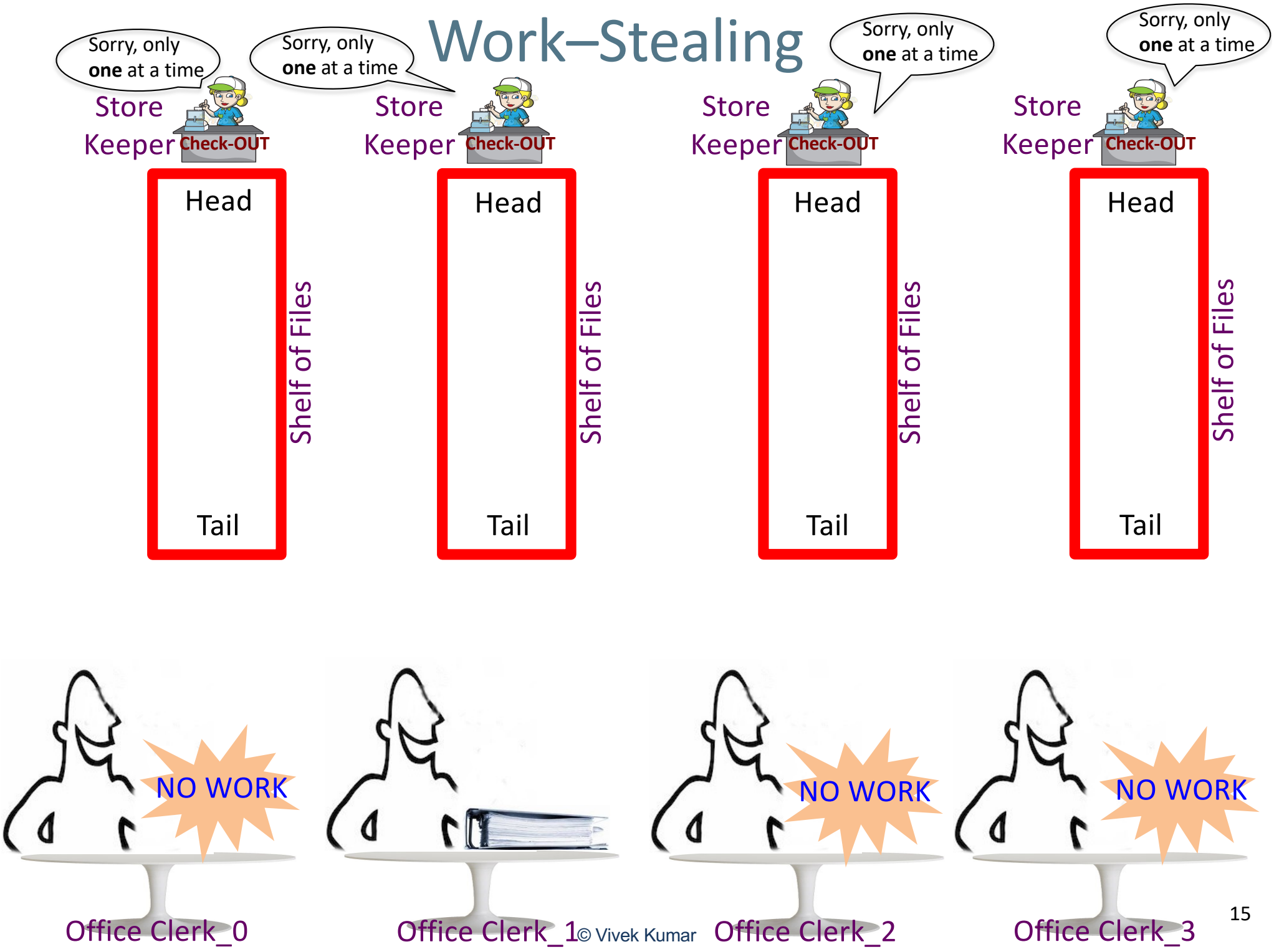
Today's Class

- Task scheduling paradigms
  - Work-sharing scheduling
  - Work-stealing scheduling

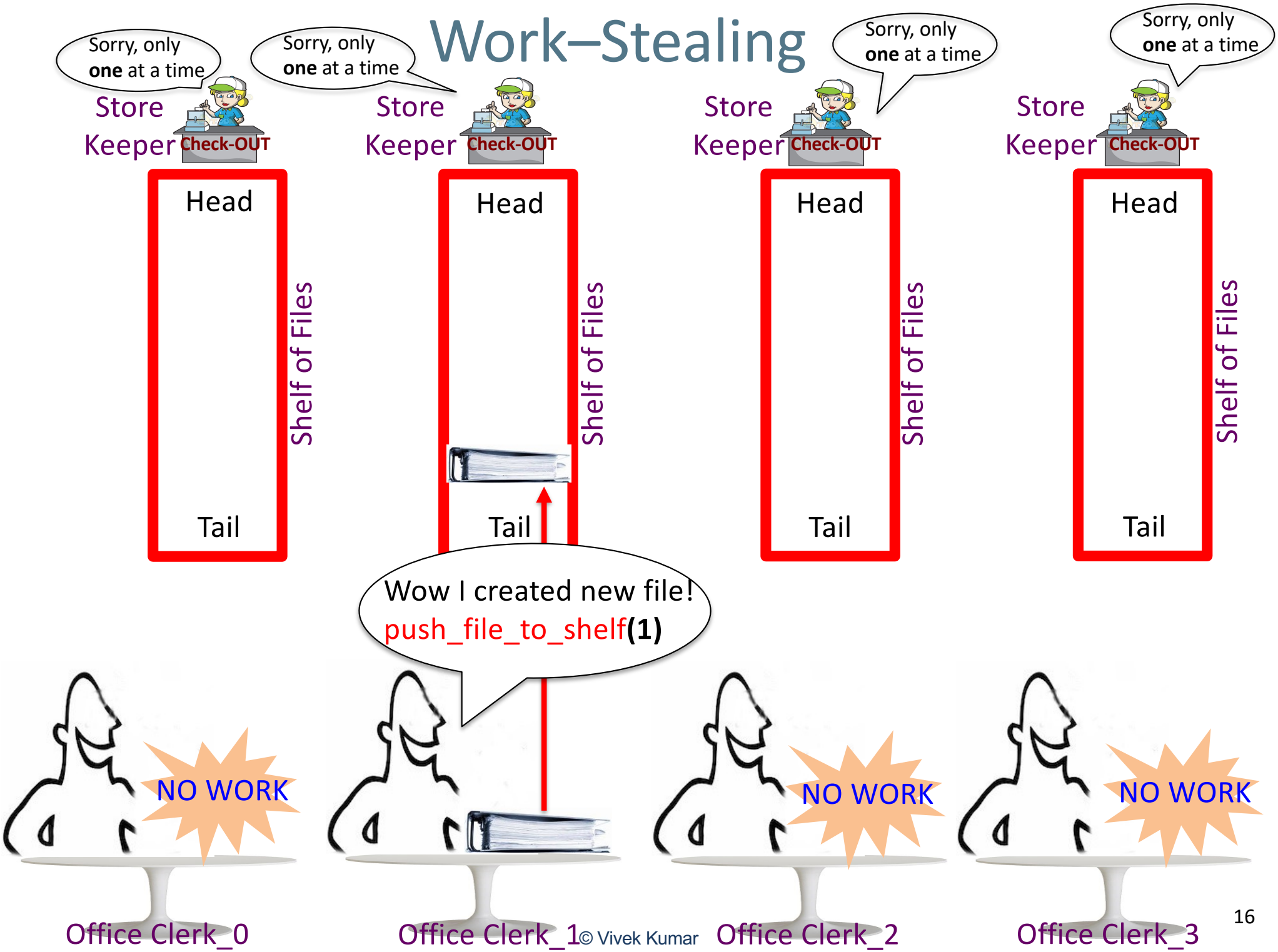
Work–Stealing



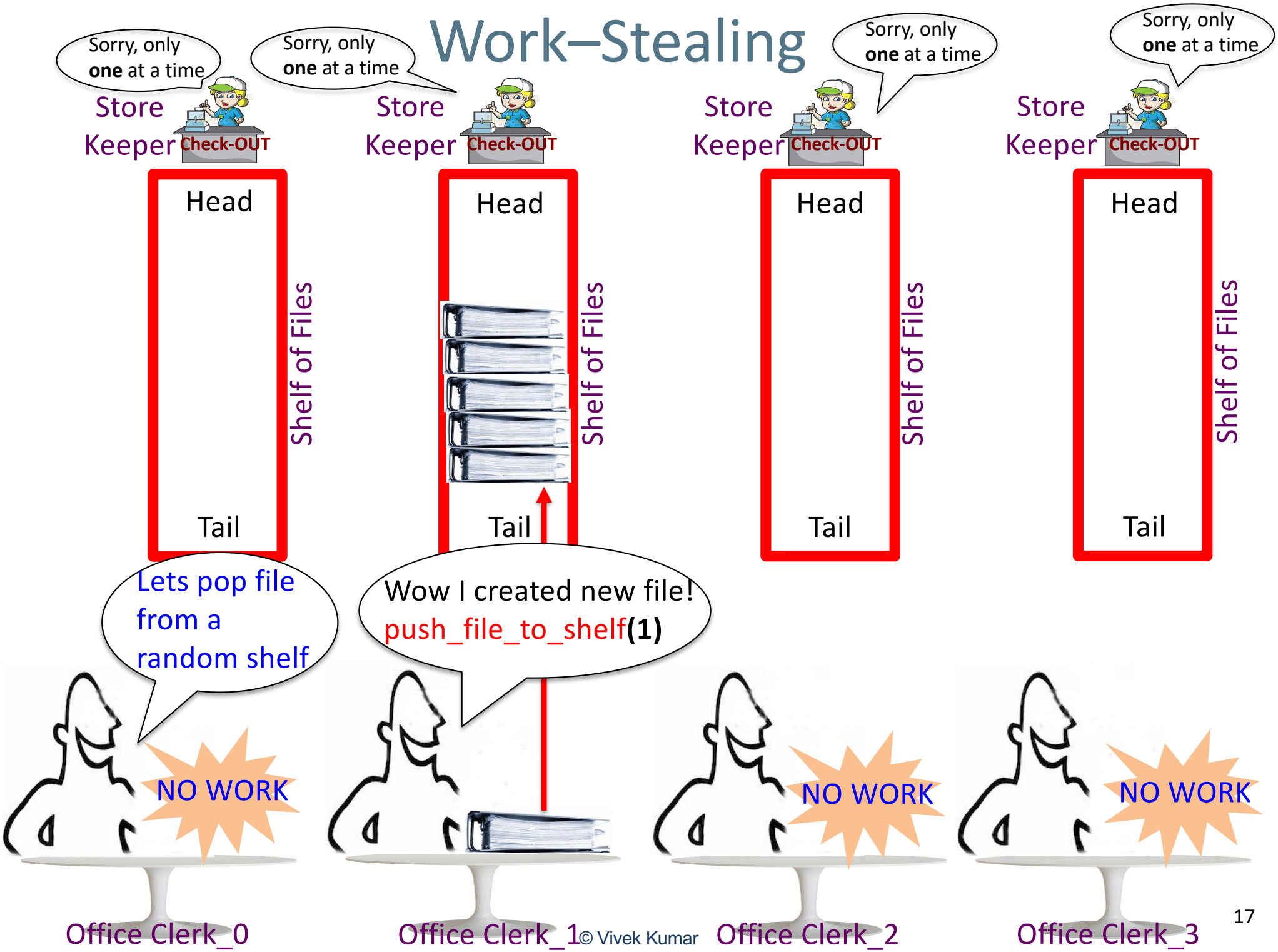
Work–Stealing



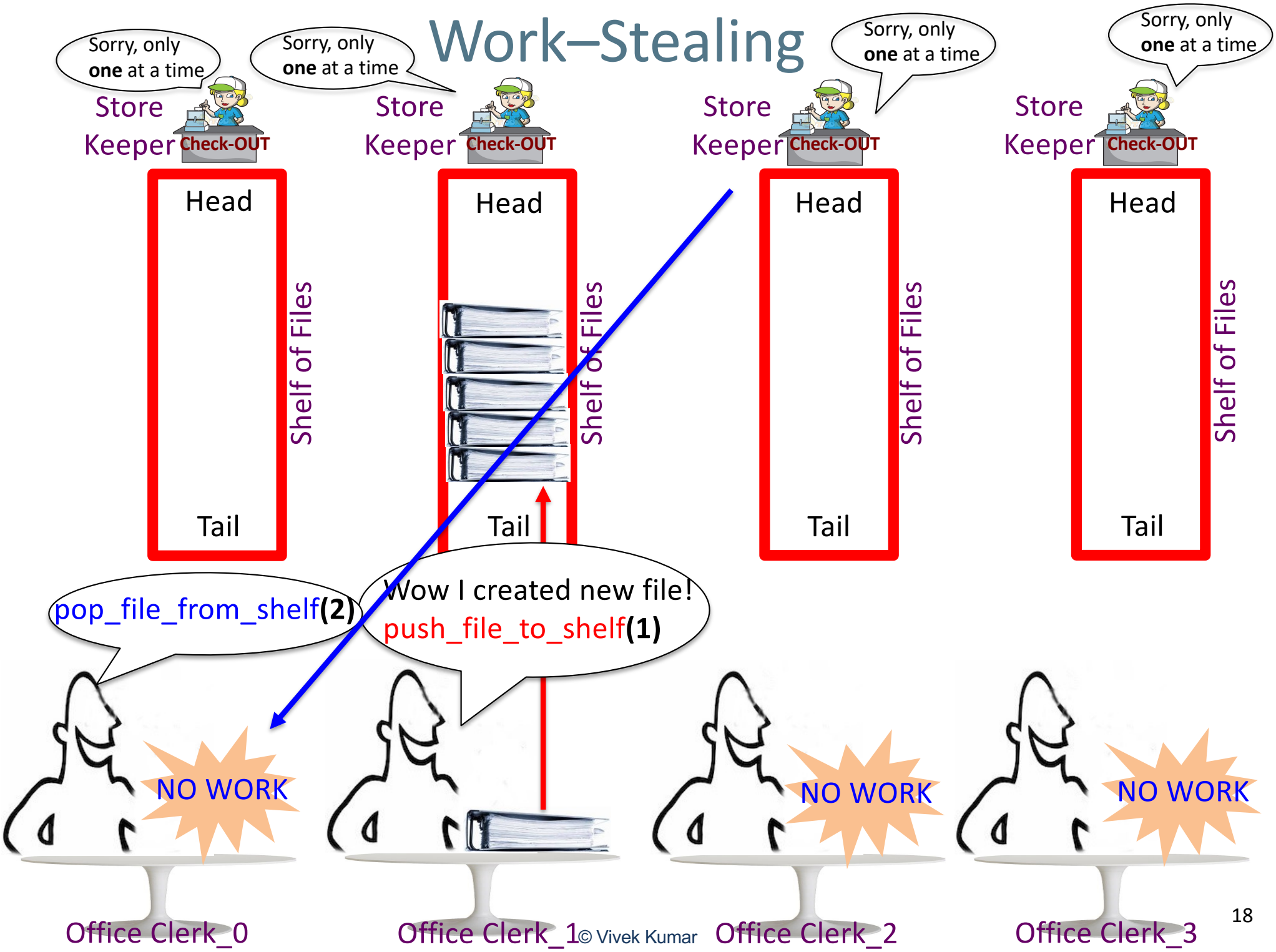
Work–Stealing



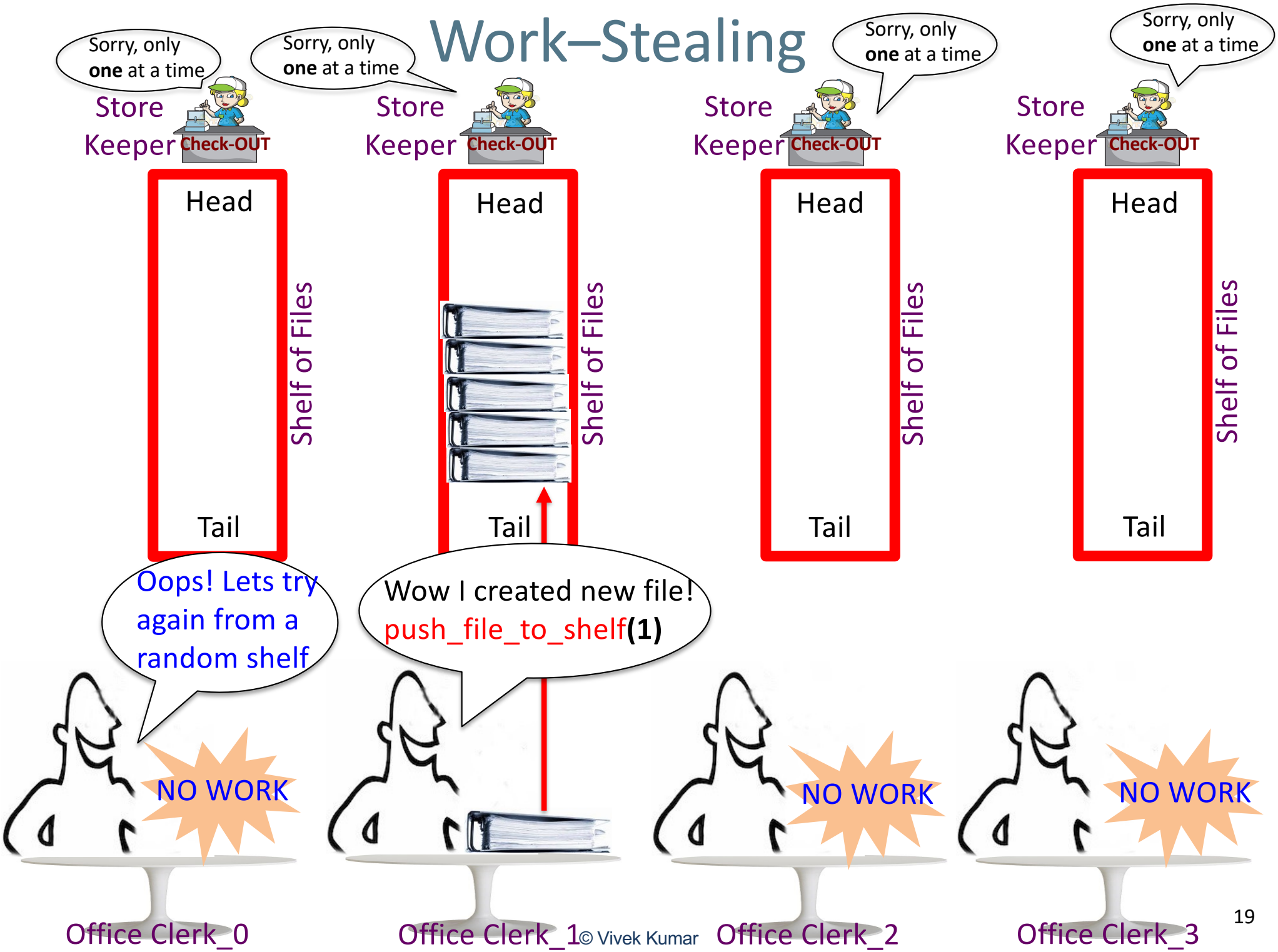
Work–Stealing



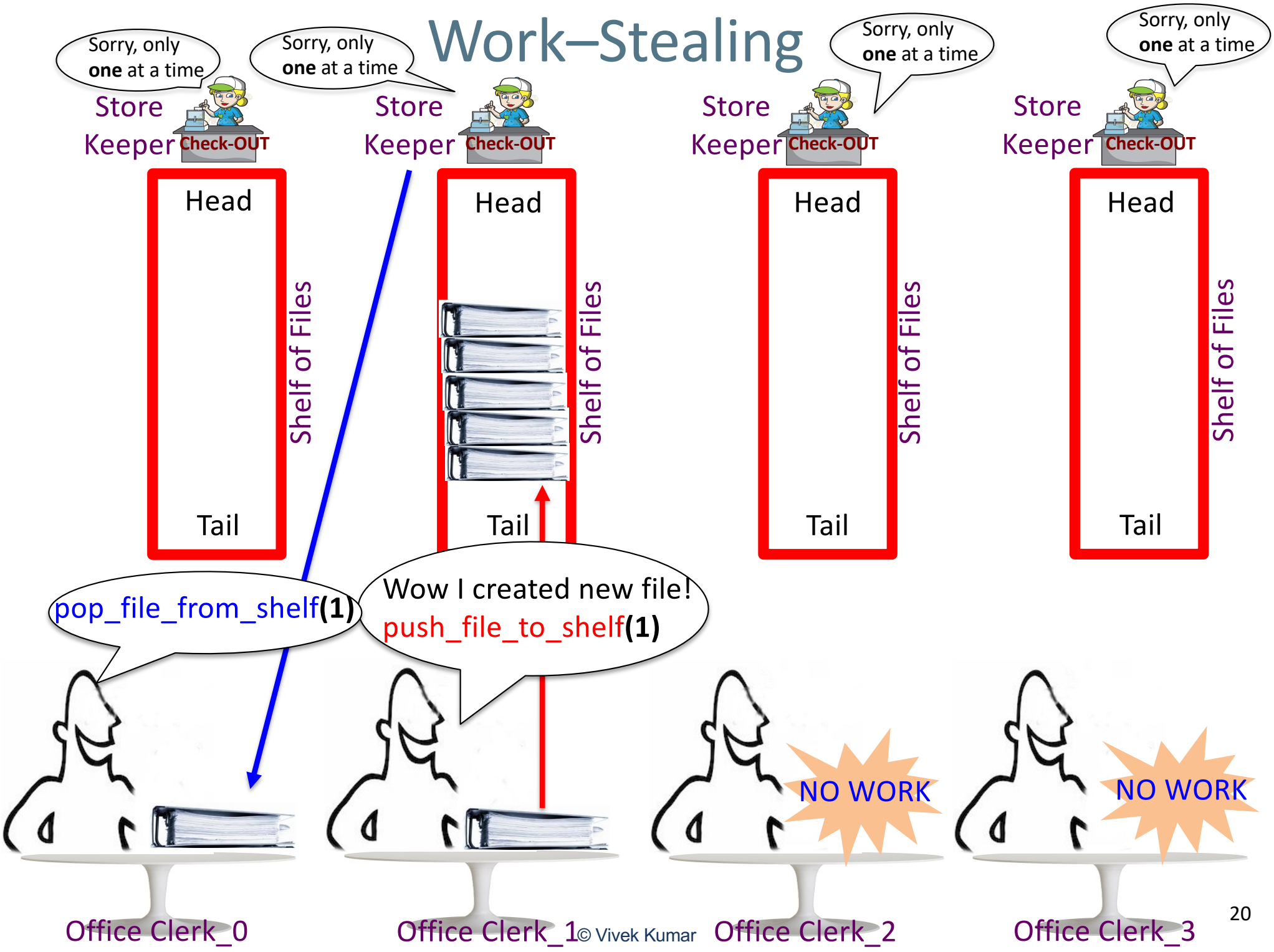
Work–Stealing



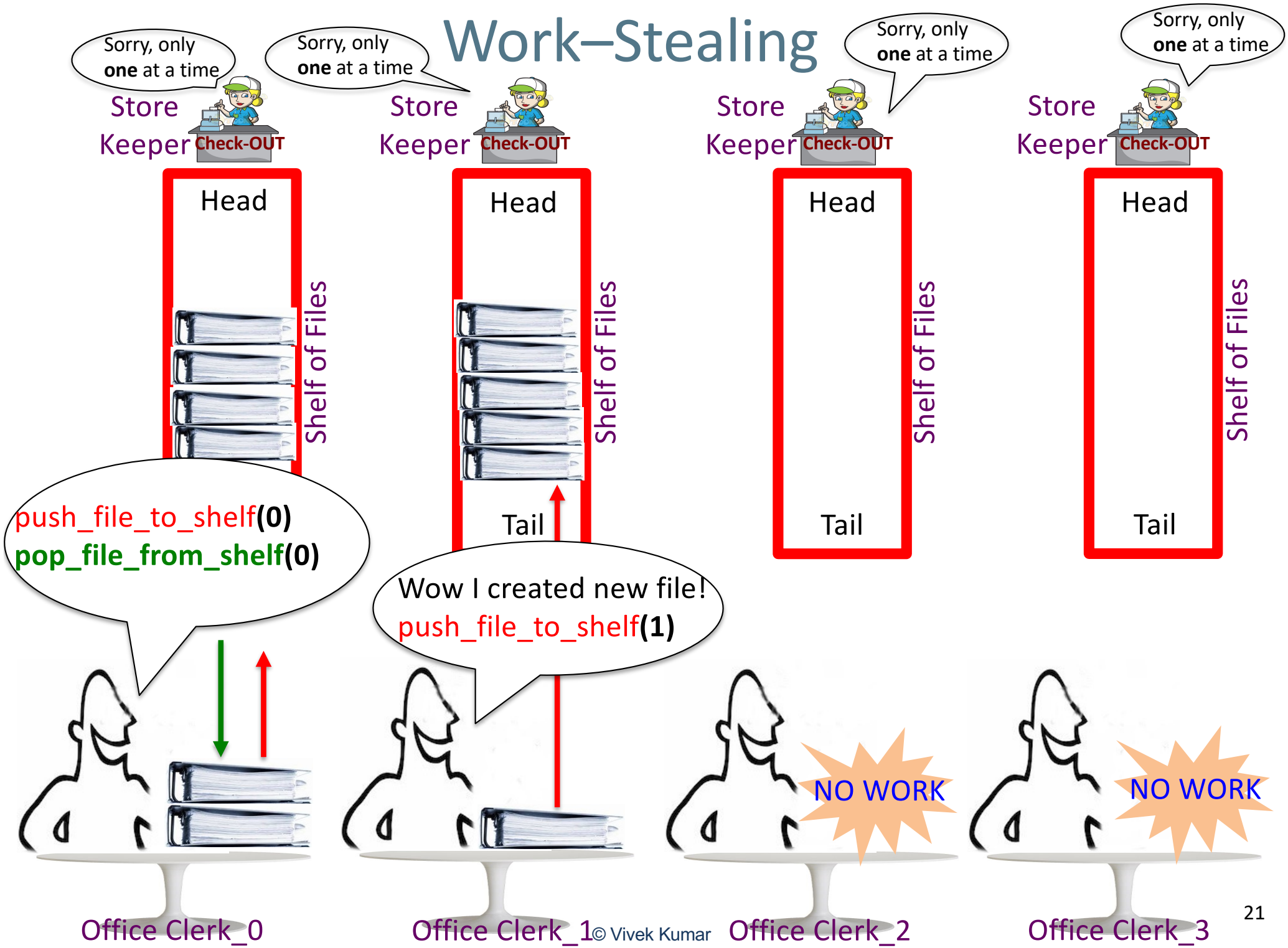
Work–Stealing



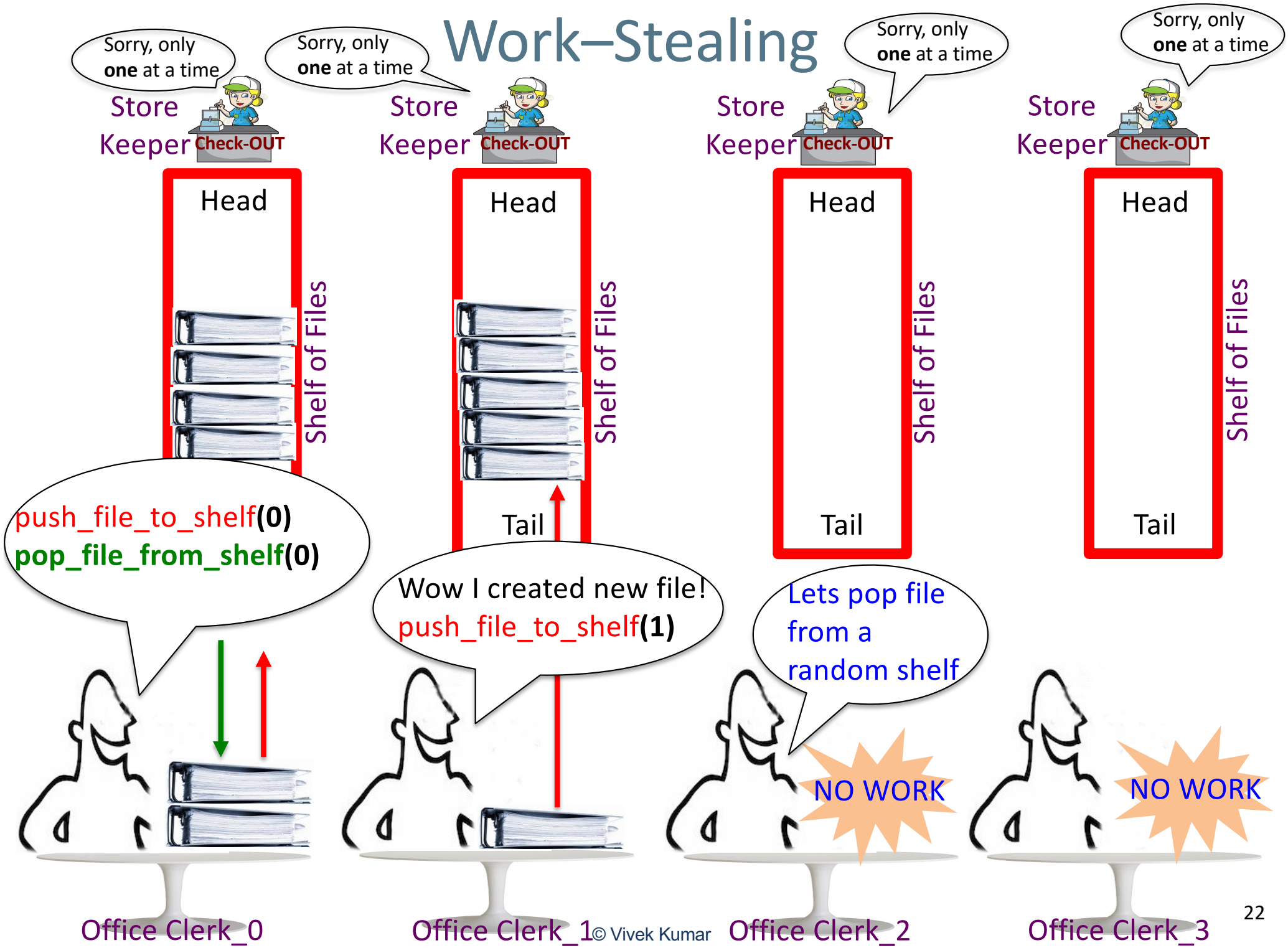
Work–Stealing



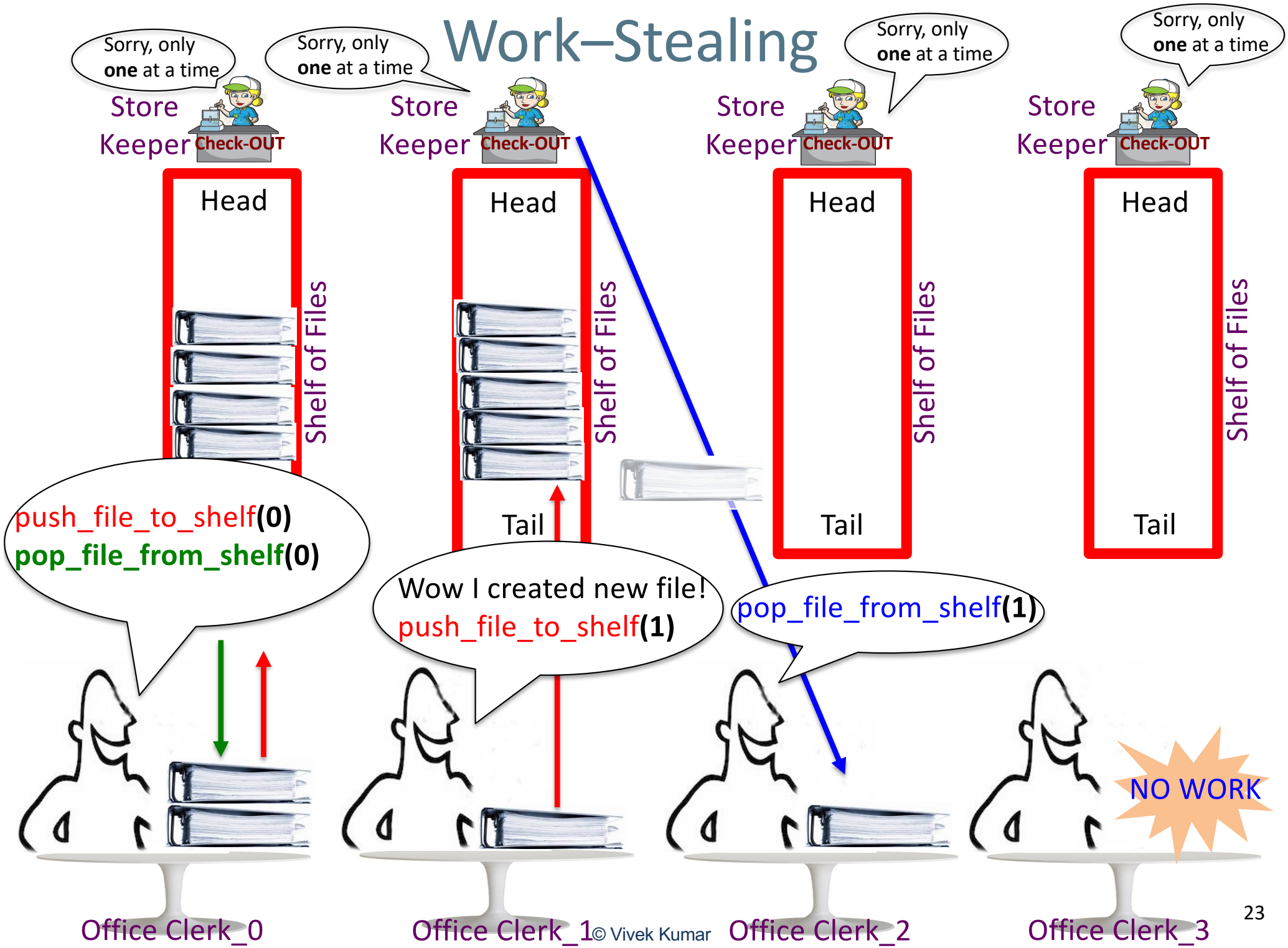
Work–Stealing



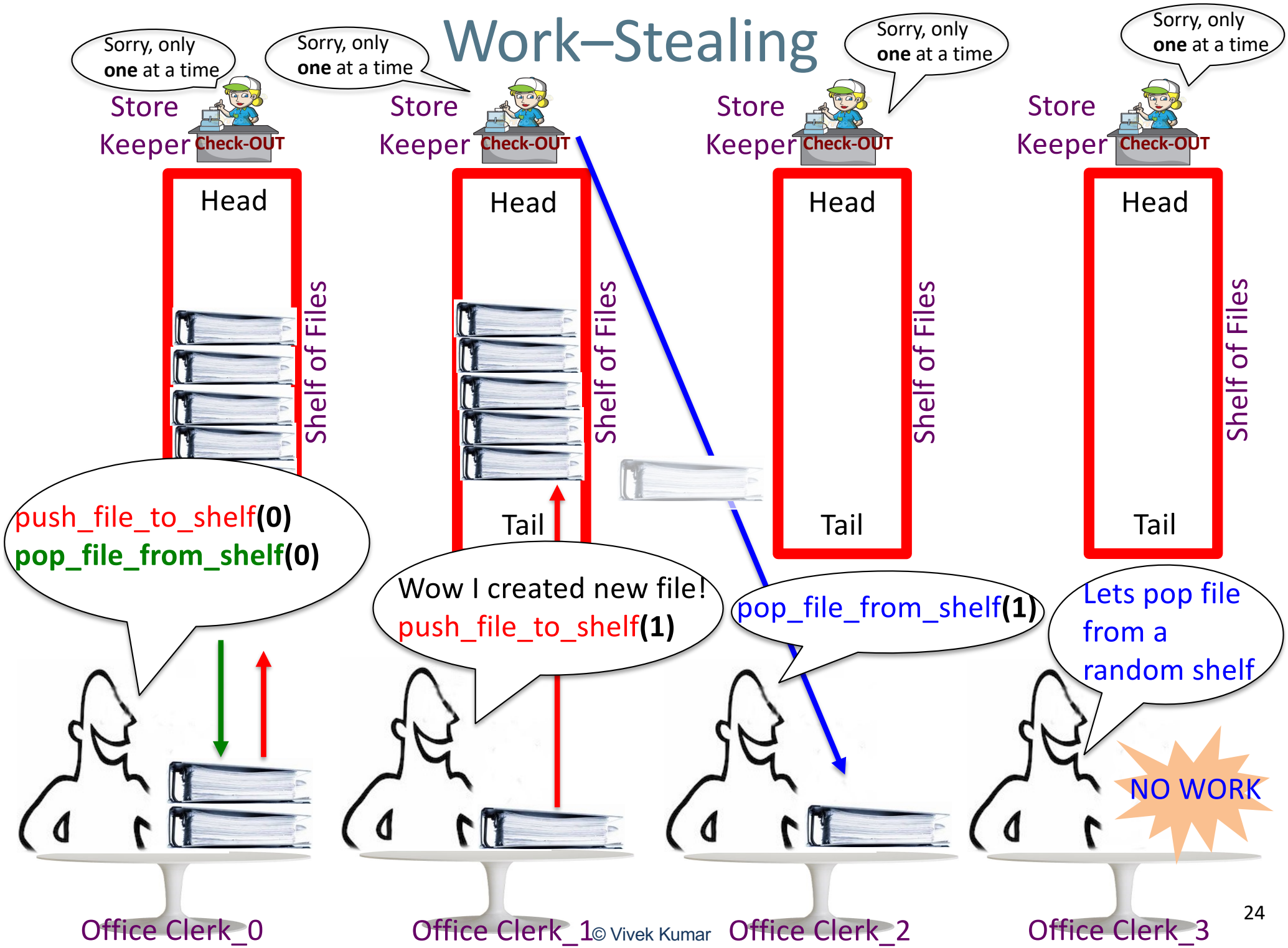
Work–Stealing



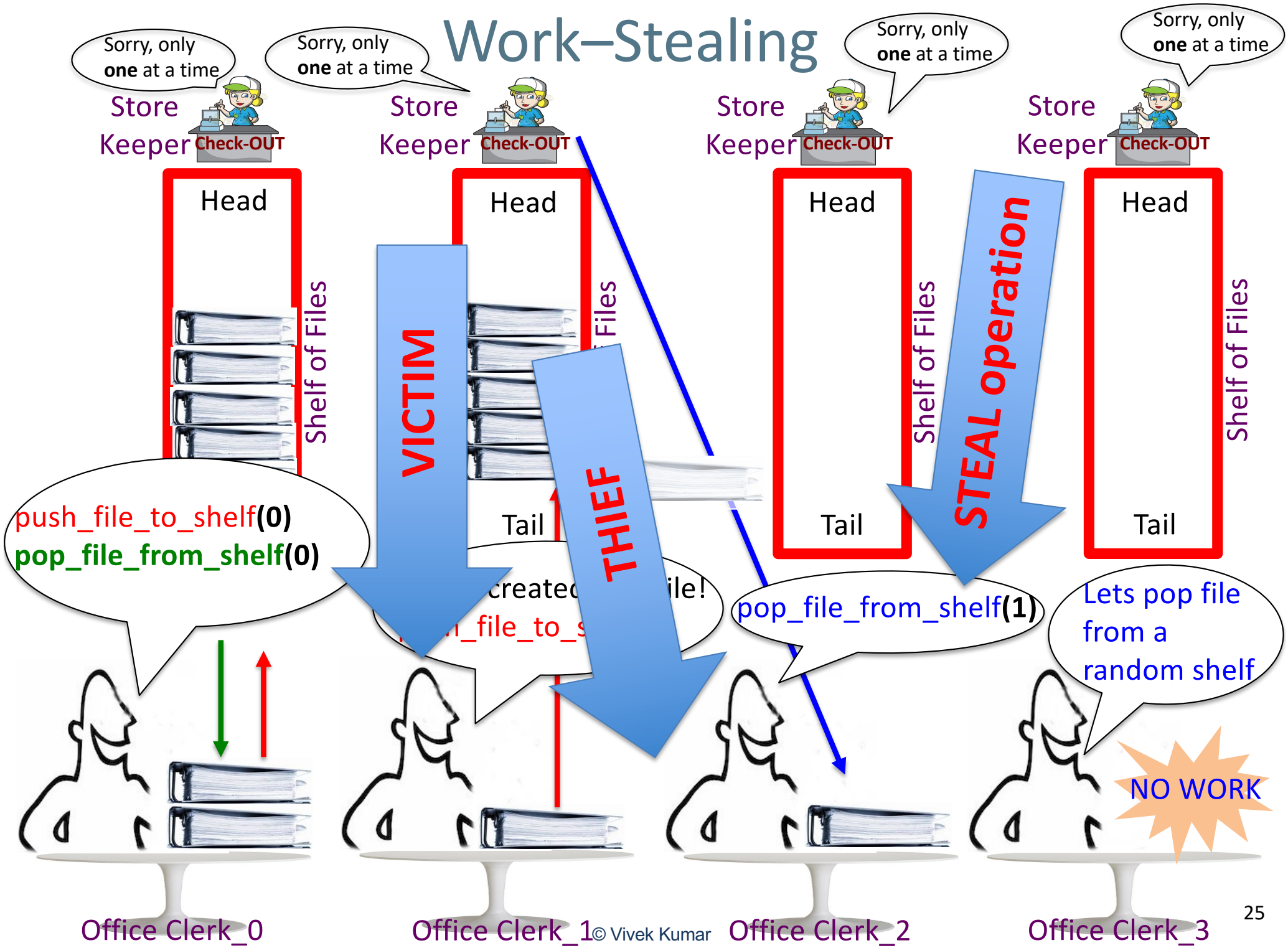
Work–Stealing



Work–Stealing



Work–Stealing



Task Scheduling Analogy With an Office

Store Keeper

Office File

Task

Office Clerk

Lock (usually implemented through compare-and-swap atomic operations, e.g., gcc built-in atomics)

Worker Thread (e.g., Pthread)

Shelf of Files in Work-stealing

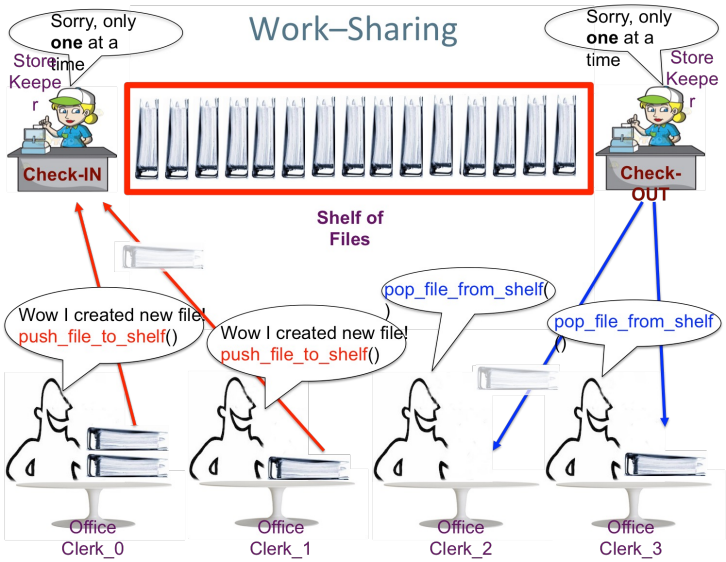
Shelf of Files in Work-sharing

Global FIFO queue

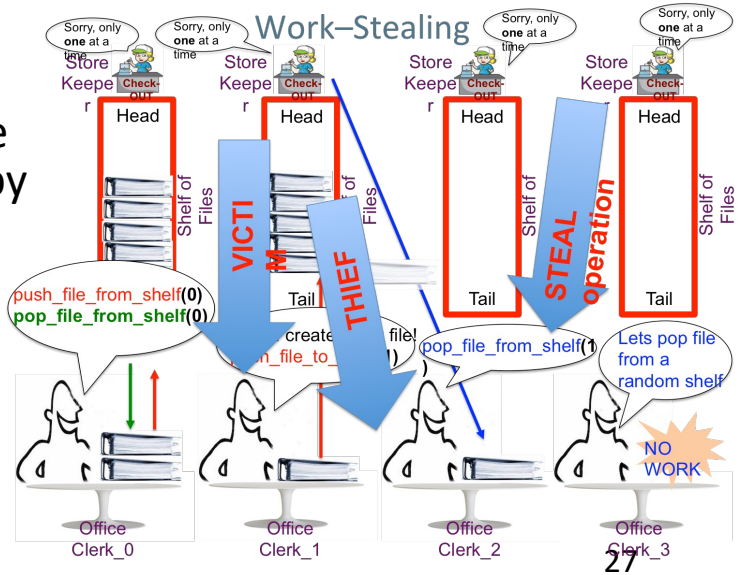
Per worker LIFO queue ("**deque**"), where the **victim push** and **pop** tasks from the **tail** and **thief steals** task from the **head**. **Pop** and **steal** are serialized on a deque only in case there is **one** task remaining

Work-Sharing v/s Work-Stealing

- Work-sharing
  - Busy worker re-distributes the task eagerly
  - Easy implementation through global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck



- Work-stealing
  - Busy worker pays little overhead to enable stealing
    - A lock is required for pop and steal only in case single task remaining on deque (only feasible by using atomic operations)
    - Idle worker steals the tasks from busy workers
  - Distributed task pools
  - Better scalability
    - NUMA?



Next Class

- Types of work-stealing
- Memoization
- Quiz-2 during next lecture (Tuesday)
  - **Syllabus**: Lectures 05-08

Reading Materials

- A Java Fork/Join framework, Doug Lea, ACM, 2000
  - http://gee.cs.oswego.edu/dl/papers/fj.pdf